VIRTUAL COMPUTER HAVING JIT COMPILER

Patent number: JP2003140909
Publication date: 2003-05-16
Inventor: SHIMURA HIROYA
Applicant: FUJITSU LTD
Classification:
- international: G06F9/45
- european:
Application number: JP20010341577 20011107
Priority number(s):

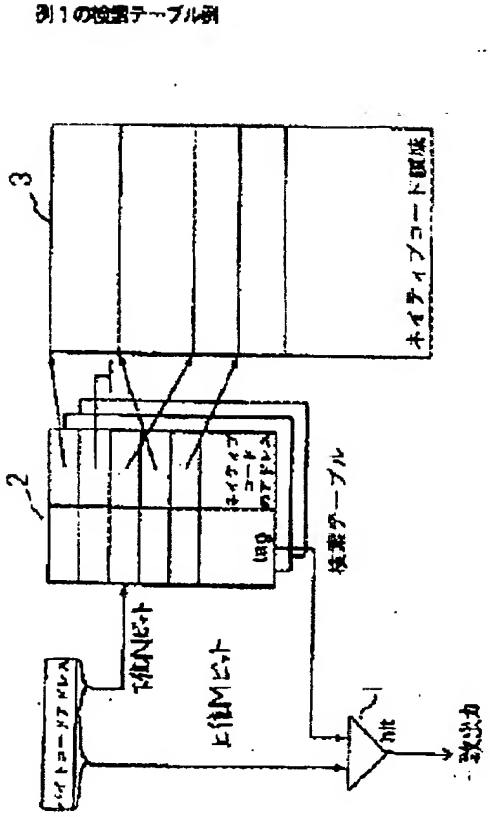
Also published as:
EP1445695 (A1)
WO03040918 (A1)
US2004210865 (A1)

Report a data error here

Abstract of **JP2003140909**

PROBLEM TO BE SOLVED: To improve responsiveness by increasing speed of processing within a limited memory capacity in relation to a virtual computer having a JIT compiler.

SOLUTION: This virtual computer comprises a function of interpreting and executing a bite code of the virtual computer with software, and a code cache as a storing region of a native code having the limited capacity on a memory. The computer also comprises the JIT compiler that compiles the bite code to a native code capable of being directly executed by the virtual computer in executing the bite code, stores the native code in the code cache 3, and then executes the native code. The computer also comprises a retrieving table 2 for retrieving information used for determining whether the native code is compiled and an address of the compiled native code based on the address of the bite code.



(19)日本国特許庁（ＪＰ）　(12)**公開特許公報**（Ａ）　(11)特許出願公開番号
特開2003－140909
（P2003－140909A）
(43)公開日　平成15年５月16日(2003.5.16)

(51)Int.Cl.⁷	識別記号	ＦＩ	ﾃｰﾏｺｰﾄﾞ(参考)
Ｇ０６Ｆ　9/45		Ｃ０６Ｆ　9/44	３２０Ｃ　５Ｂ０８１

審査請求　未請求　請求項の数５　ＯＬ　（全 17 頁）

(21)出願番号　特願2001－341577(P2001－341577)

(22)出願日　平成13年11月７日(2001.11.7)

(71)出願人　000005223
富士通株式会社
神奈川県川崎市中原区上小田中４丁目１番１号

(72)発明者　志村　浩也
神奈川県川崎市中原区上小田中４丁目１番１号　富士通株式会社内

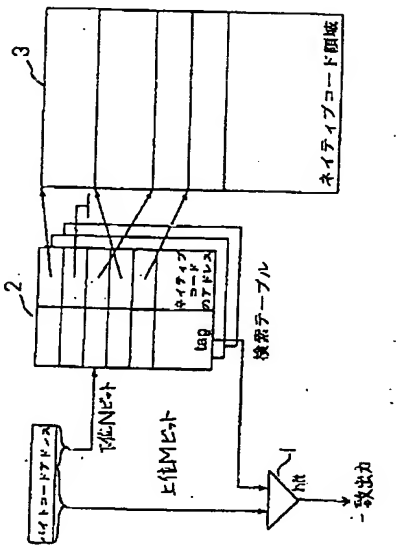
(74)代理人　100096530
弁理士　今村　辰夫　（外２名）

Ｆターム(参考)　5B081 AA09 CC41 DD01

(54)【発明の名称】　ＪＩＴコンパイラを備えた仮想計算機

(57)【要約】
【課題】本発明はＪＩＴコンパイラを備えた仮想計算機に関し、限られたメモリ容量で処理の高速化を図り、応答性能の向上を図る。
【解決手段】仮想計算機のバイトコードをソフトウェアで解釈・実行する機能を有し、メモリ上に容量の制限されたネイティブコードの格納領域としてコードキャッシュを持ち、バイトコードを実行する際にバイトコードを仮想計算機が直接実行できるネイティブコードにコンパイルしてコードキャッシュ３に格納した後、そのネイティブコードを実行するＪＩＴコンパイラを備え、バイトコードのアドレスから、コンパイルされているか否かを判断するための情報と、コンパイルされたネイティブコードのアドレスを検索する検索テーブル２を備えた。

例１の検索テーブル例



理である。

【００１０】前記インタプリタの処理速度が遅いことを解決する方法として、ＪＩＴ（Just-IN-Compiler）コンパイラを使う手法が一般に使用されていた。このＪＩＴコンパイラはバイトコードをネイティブコードにコンパイルする。これにより、アプリケーションは直接そのマシンのプロセッサで実行できるようになり、高速な実行が可能になる。

【００１１】一般に、ＪＩＴコンパイラは、ダウンロードしたクラス、又は実行を開始したメソッドという単位でコンパイルを行う。すなわち、メソッドのバイトコード全体をコンパイルする。また、コンパイルしたネイティブコードは、メモリ（プログラム上の格納領域）上に格納され、再び実行を行う時は再利用され、１度目の実行時には、コンパイルのオーバヘッドがあるが、２度目からはこのオーバヘッド無しに高速に実行できる。具体的には次のようになる。

【００１２】先ず、バイトコードの命令フェッチを行い（Ｓ１１）、フェッチした命令をデコードする（Ｓ１２）。そして、デコードした結果が実行であれば、その命令を実行し（Ｓ１３、Ｓ１４）、Ｓ１１の処理へ移行する。また、デコードした結果が分岐であれば、分岐処理を行い（Ｓ１５）、Ｓ１１の処理へ移行する。更に、デコードした結果がメソッド呼び出しであれば、メソッド呼び出しを行い（Ｓ１６）、コンパイル済みか否かを判断する（Ｓ１７）。

【００１３】その結果、コンパイル済みでなければ、コンパイルすべきか否かを判断し（Ｓ１８）、コンパイルすべきでなければ、Ｓ１１の処理へ移行するが、コンパイルすべきであれば、メソッドで呼び出したコードをＪＩＴコンパイラによりコンパイルしてネイティブコードを生成し、ネイティブコードの格納領域に格納し（Ｓ１９）、ネイティブコードを実行する（Ｓ２０）。ネイティブコードの実行中、メソッド呼び出しがあれば、Ｓ１６の処理へ移行し、メソッドの処理が終了すれば、Ｓ１１へ移行する。また、Ｓ１７の処理において、コンパイル済みであれば、Ｓ２０の処理へ移行する。以降、同様にして処理を行う。

【００１４】なお、前記バイトコード（bytecode）とは、Ｊａｖａで開発したソフトのバイナリ表現形式として米サン・マイクロシステムズが定めたＪａｖａ仮想マシン（仮想計算機）のマシン・コードのことを言う。また、バイトコードをマシンが直接実行できるネイティブコードへ変換するソフトを「ＪＩＴ（Just-in-time）コンパイラ」と呼ぶ。

【００１５】また、「メソッド」（Method）とは、オブジェクト指向プログラミングにおいて、オブジェクトの実行する操作の方法を記述したプログラム（サブルーチンと同じ意味）のことを言う。

【００１６】

【発明が解決しようとする課題】前記のような従来のものにおいては、次のような課題があった。

【００１７】すなわち、携帯電話機やＰＤＡ等の携帯型無線通信機器では使用できるメモリ容量が限られており、コンパイルしたコードを全てメモリ上に格納することはできない。また、前記携帯型無線通信機器でのソフトウェアの実行は、ゲームなどの対話型のアプリケーションが多く、応答性を高めるためには、コンパイルにあまり時間をかけることはできない。

【００１８】特に、前記携帯型無線通信機器に搭載されたプロセッサでは処理能力が比較的低いため、大きなプログラムをコンパイルすると、実行中のプログラムが一瞬止まって見えることもある。また、一度しか実行されないようなバイトコードの場合、コンパイルする時間のオーバーヘッドのため、インタプリタで実行するよりも遅くなる。

【００１９】本発明は、このような従来の課題を解決し、限られたメモリ容量で処理の高速化を図り、応答性能の向上を図ることを目的とする。

【００２０】

【課題を解決するための手段】本発明は前記の目的を達成するため、次のように構成した。

【００２１】(1)：仮想計算機のバイトコードをソフトウェアで解釈・実行する機能を有し、メモリ上に、容量の制限されたネイティブコードの格納領域としてコードキャッシュを持ち、前記バイトコードを実行する際に、ＪＩＴコンパイラによりバイトコードを仮想計算機が直接実行できるネイティブコードにコンパイルして前記コードキャッシュに格納した後、そのネイティブコードを実行するＪＩＴコンパイラを備えた仮想計算機において、前記バイトコードのアドレスから、コンパイルされているか否かを判断するための情報と、コンパイルされたネイティブコードのアドレスを検索する検索テーブルを備えていることを特徴とする。

【００２２】(2)：前記(1)の仮想計算機において、前記検索テーブル上、又は前記コードキャッシュ上に、命令の実行回数を計数するための実行カウンタを設け、バイトコードを実行する際に、前記実行カウンタの値を調べ、予め決めた特定回数より小さい場合はインタプリタで実行し、前記特定回数以上の場合は、ＪＩＴコンパイラによりバイトコードをコンパイルしてから実行する選択処理手段を備えていることを特徴とする。

【００２３】(3)：前記(1)の仮想計算機において、前記検索テーブルに、１度目の実行時にバイトコードのアドレスだけ登録し、ネイティブコードのアドレスの登録は特別な値、又は数字の０を登録にしておく検索テーブル情報登録手段と、１度目はインタプリタで実行し、２度目はＪＩＴコンパイルによりコンパイルして実行する実行制御手段を備えていることを特徴とする。

【００２４】(4)：前記(1)の仮想計算機において、コ

パイラ（以下、単に「コンパイラ」とも記す）を備えた仮想計算機に関するものであり、バイトコードのアドレスから、コンパイルされているか否かを判断するための情報（後述する）と、コンパイルされたネイティブコードのアドレスを検索する検索テーブルを備えている。

【００４０】そして、コンパイルを行う時にネイティブコードの格納領域に空きがなければ、既に格納済みのコンパイルコードを破棄して前記格納領域に空き領域を作り、検索テーブルを更新するものである。

【００４１】②：フローチャートによる処理（その１）の説明

図１は例１の処理フローチャート（その１）である。以下、図１に基づいて、例１の処理（その１）を説明する。なお、Ｓ３１～Ｓ４０は各処理ステップを示す。この処理では、全てのバイトコードを対象としてコンパイルされているかどうかをチェックする例であり、仮想計算機上の処理である。

【００４２】先ず、全てのバイトコードに対して、検索テーブル上のタグ（ｔａｇ）情報を利用して（詳細は後述する）コンパイル済みか否かを判断し（Ｓ３１）、コンパイル済みでなければ、コンパイルすべきか否かを判断し（Ｓ３２）、コンパイルすべきでないと判断した場合には、バイトコード命令のフェッチを行い（Ｓ３３）、フェッチされた命令のデコードを行う（Ｓ３４）。

【００４３】その結果、実行であれば（Ｓ３５、Ｓ３６）その命令を実行し、Ｓ３１の処理へ移行する。また、分岐であれば（Ｓ３７）分岐処理を行い、Ｓ３１の処理へ移行する。更に、メソッド呼び出しであれば（Ｓ３８）、メソッド呼び出しを行い、Ｓ３１の処理へ移行する。

【００４４】また、前記Ｓ３２の処理で、コンパイルすべきであると判断した場合には、部分的にコンパイルし、ネイティブコードを生成してネイティブコード格納領域へ格納し（Ｓ３９）、ネイティブコードを実行し（Ｓ４０）、Ｓ３１の処理へ移行する。また、Ｓ３１の処理で、コンパイル済みであれば、Ｓ４０の処理へ移行する。前記処理において、Ｓ３１～Ｓ３８はインタプリタの処理であり、Ｓ３９、Ｓ４０がＪＩＴコンパイラによる処理である。

【００４５】③：フローチャートによる処理（その２）の説明

図２は例１の処理フローチャート（その２）である。以下、図２に基づいて、例１の処理（その２）を説明する。なお、Ｓ４１～Ｓ５０は各処理ステップを示す。

【００４６】この処理では、分岐、メソッド呼び出しなどの実行制御が途切れる地点で、コンパイルのチェックを行う。一般のＪＩＴコンパイラとの違いとしては、コンパイルをメソッド単位で行わず、以下に説明する例５、例６、例７、例８等の条件で、コンパイルを中断することが特徴である。

【００４７】先ず、バイトコード命令のフェッチを行い（Ｓ４１）、命令デコードを行う（Ｓ４２）。その結果、実行であれば（Ｓ４３、Ｓ４４）、その命令を実行し、Ｓ４１の処理へ移行する。また、分岐であれば（Ｓ４５）、分岐処理を行い（Ｓ４５）、メソッド呼び出しであれば（Ｓ４６）メソッド呼び出しを行う。

【００４８】そして、分岐又はメソッド呼び出しを行った場合、コンパイル済みか否かを判断し（Ｓ４７）、コンパイル済みでなければ、コンパイルすべきか否かを判断する（Ｓ４８）。その結果、コンパイルすべきでないと判断した場合には、Ｓ４１の処理へ移行する。

【００４９】また、コンパイルすべきであると判断した場合は、途中までコンパイルを行い、ネイティブコードを生成しネイティブコード格納領域へ格納する（Ｓ４９）。そして、ネイティブコードを実行し（Ｓ５０）、Ｓ４７の処理、又はＳ４６の処理へ移行する。

【００５０】④：検索テーブルの説明

図３は例１の検索テーブル例である。前記のようにコンパイル済みか否かを調べて判断するために、検索テーブルを使用するが、その検索テーブルの１例を図３に示す。図３において、１はコンパレータ、２は検索テーブル、３はコードキャッシュである。

【００５１】この検索テーブル２は、タグ（ｔａｇ）と、ネイティブコードのアドレス（コードキャッシュ３内のネイティブコード領域のアドレス）との対応情報が登録できるようになっており、ネイティブコードのアドレスからコードキャッシュ３のネイティブコード領域のデータが検索できるようになっている。

【００５２】また、検索テーブル２は、バイトコードアドレス（上位Ｍビット＋下位Ｎビットとする）の下位Ｎビットを利用してタグ（ｔａｇ）を検索することで、ネイティブコードのアドレスが検索できるようになっている。そして、バイトコードアドレスの上位Ｍビットをコンパレータ１に入力すると共に、検索テーブルのタグ（ｔａｇ）情報をコンパレータ１に入力し、両者が一致するか否かの比較を自動的に行うようになっている。

【００５３】前記比較の結果、両者が一致（ｈｉｔ）した場合、コンパレータ１から一致出力が出され、この一致出力信号によりコンパイル済みと判断できるようになっている。

【００５４】⑤：例１の特徴

前記図３の検索テーブルを検索することでコンパイルされているか否かを判断したり、コンパイルされている場合に、該当するネイティブコードの格納アドレス（コードキャッシュ３内のネイティブコード領域のアドレス）を検索することが容易になる。

【００５５】その結果、バイトコードをネイティブコードにコンパイルすることで、アプリケーションプログラムの実行時間を速くすることが可能になる。また、従来

【００６９】また、Ｓ８１の処理で、テーブル登録済みであると判断した場合（２度目）は、ＪＩＴコンパイラによる処理を行う。この場合、ネイティブコードアドレスへジャンプし（Ｓ９１）、エントリＢからＪＩＴコンパイラによるコンパイルを行い（Ｓ９２）、ネイティブコードアドレス登録を行う（Ｓ９３）。そして、ネイティブコードを実行し（Ｓ９４）、Ｓ８１の処理へ移行する。

【００７０】また、前記Ｓ９１の処理結果により、Ｓ９４に移行してネイティブコードを実行し（Ｓ９４）、Ｓ８１の処理へ移行する。

【００７１】❸：例３の特徴

コンパイル処理は時間を要するため、一度しか実行はないようなバイトコードはコンパイルして実行するよりもインタプリタで実行する方が速い。そこで、前記のように処理を行うことで、アプリケーションプログラムの全体の実行速度を速くすることができる。また、前記例２と比較して、実行カウンタのための領域が必要なく、メモリを節約できる。

【００７２】(4) ：例４の詳細な説明

①：例４の内容

例４は、例１において、検索テーブルはコンパイルされているかどうかだけの情報（タグの情報）を持ち、ネイティブコードのアドレスはバイトコードのアドレスから計算する例である。

【００７３】❷：検索テーブルとコードキャッシュの構成例

図７は例４の検索テーブルとコードキャッシュの構成例である。この検索テーブル２には、タグ（ｔａｇ）のみ格納されている。この場合、バイトコードアドレスの下位Ｎビットを利用して検索テーブル２をアクセスすると共に、前記Ｎビットの下位アドレスを演算器４により整数倍、例えば、８倍（×８）した値によりコードキャッシュ３のネイティブコード領域のアドレスへアクセスできるようになっている。

【００７４】また、検索テーブル２は、バイトコードアドレス（上位Ｍビット＋下位Ｎビットとする）の下位Ｎビットを利用してタグ（ｔａｇ）を検索できるようになっている。そして、バイトコードアドレスの上位Ｍビットをコンパレータ１に入力すると共に、検索テーブルのタグ（ｔａｇ）情報をコンパレータ１に入力し、両者が一致するか否かの比較を行う。前記比較の結果、両者が一致した場合、コンパレータ１から一致出力が出されるが、この一致出力信号によりコンパイル済みと判断する。

【００７５】❸：例４の特徴

バイトコードのアドレスに対し、コンパイルされるネイティブコードのメモリ上の位置が一意に決まるため、テーブルからネイティブコードのアドレスを索引する必要がなくなる。また、置換の際には、破棄すべきネイティブコードを探す必要がなく単純に処理できる。

【００７６】(5) ：例５の詳細な説明

①：例５の内容

例５は、例１において、コンパイル時間を短縮（応答性を向上）するために、コンパイルする範囲を制限する。処理の途中であっても、バイトコードの特定の命令（分岐命令、メソッド呼び出しなど）を検出したら、そこまでをコンパイルし、処理を中断する命令（ネイティブコードの分岐命令や割り込み命令）を生成し（ネイティブコードの最終コードの後に、分岐命令や割り込み命令を生成し）、残りはコンパイルしない例である。

【００７７】すなわち、仮想計算機において、コンパイル時間を短縮するために、コンパイルする範囲を制限し、処理の途中であっても、バイトコードの特定の命令を検出したら、そこまでをコンパイルして処理を中断する命令を生成し、残りはコンパイルしないように処理を制限する処理制限手段を備えている。

【００７８】❷：例５の特徴

必ず実行する部分だけがコンパイルされ効率が良い。例えば、“ＩＦ条件ＴＨＥＮ処理”のようなバイトコードの列をコンパイルするとき、「条件」を処理する部分だけコンパイルを行い、「処理」の部分はコンパイルしない。

【００７９】この場合、従来は、メソッド全体をコンパイルしていたので、全く実行しない部分もコンパイルしていた。これに対して例５では、必ず実行する部分だけをコンパイルするので、コンパイルに消費する時間の無駄がなく高速である。

【００８０】また、従来は、メソッド全体をコンパイルしていたので、全く実行しない部分のネイティブコードも生成していた。しかし、例５では、必ず実行する部分だけをコンパイルするので、生成するネイティブコードに無駄がなく、メモリの消費が抑えられる。また、アプリケーションプログラムの応答性を向上させることができる。すなわち、コンパイルしている最中はアプリケーションプログラムの実行は停止するので、この停止している時間が短縮する。

【００８１】(6) ：例６の詳細な説明

①：例６の内容

例６は、コンパイル時間の短縮を行う例であり、例５において、特定な命令の検出のほかに、バイトコードの量でコンパイルを中断する。例えば、１００命令コンパイルしたところでコンパイルを中断する例である。なお、例６は、例４と同様に、例１の処理におけるコンパイルを中断させる条件を提供するものである。

【００８２】❷：例６の特徴

例５と同じように、必ず実行する部分だけをコンパイルするので、生成するネイティブコードに無駄がなく、メモリの消費が抑えられる。また、アプリケーションプログラムの応答性を向上させることができる。すなわち、

コンパイルしている最中はアプリケーションプログラムの実行は停止するので、この停止している時間が短縮する。

【００８３】(7) ：例７の詳細な説明

①：例７の内容

例７はコンパイル時間の短縮を行う例であり、例５において、特定な命令の検出のほかに、生成したネイティブコードの量でコンパイルを中断する。例えば、コンパイルが１２８ワード分だけネイティブコードを生成したら、コンパイルを中断する。なお、例７は、例４と同様に、例１の処理におけるコンパイルを中断させる条件を提供するものである。

【００８４】②：例７の特徴

例５と同じように、必ず実行する部分だけをコンパイルするので、生成するネイティブコードに無駄がなく、メモリの消費が抑えられる。また、アプリケーションプログラムの応答性を向上させることができる。すなわち、コンパイルしている最中はアプリケーションプログラムの実行は停止するので、この停止している時間が短縮する。

【００８５】(8) ：例８の詳細な説明

①：例８の内容

例８はコンパイル時間の短縮を行う例であり、特定な命令の検出のほかに、時間によってコンパイルを中断する。例えば、コンパイラがコンパイルの処理に１００ミリセカンド消費したら、コンパイルを中断する。なお、例８は、例４と同様に、例１の処理におけるコンパイルを中断させる条件を提供するものである。

【００８６】②：例８の特徴

例５と同じように、必ず実行する部分だけをコンパイルするので、生成するネイティブコードに無駄がなく、メモリの消費が抑えられる。また、アプリケーションプログラムの応答性を向上させることができる。すなわち、コンパイルしている最中はアプリケーションプログラムの実行は停止するので、この停止している時間が短縮する。

【００８７】(9) ：例９の詳細な説明

①：例９の内容

例９は、コードキャッシュの置換を行う例であり、例１において、コードキャッシュに空きがなくなったとき、コードキャッシュの先頭から順にネイティブコードを破棄する。この処理では、ＦＩＬＯ（First In Last Out）方式で最も昔にコンパイルしたコードから順に破棄する。

【００８８】②：コードキャッシュの置換の説明

図８は例９のコードキャッシュの置換処理説明図である。この例では、初期値として、Ｐ１：命令生成ポインタはコードキャッシュの先頭、Ｐ２：解放ポインタはコードキャッシュの末尾を指しておく。実行が進む（コンパイルが何度も行われる）につれて、コードキャッシュ３は小さなブロックに分けられ、ブロックの先頭にタグ（ｔａｇ）ポインタ（検索テーブルへのポインタ）と、ネクスト（ｎｅｘｔ）ポインタ（次のブロックを指すポインタ）が格納される。

【００８９】ブロックの残り部分はネイティブコードを格納する命令領域である。ネクスト（ｎｅｘｔ）ポインタはポインタである必要はなく、ブロックの切れ目が判れば良い。

【００９０】③：フローチャートによる処理の説明

図９は、例９のコードキャッシュ置換処理フローチャートである。以下、図９に基づいて、例９の処理を説明する。なお、Ｓ１０１～Ｓ１１１は各処理ステップを示す。

【００９１】この処理は、コンパイルの処理の最中にコードキャッシュ３の置換処理を行う例である。先ず、生成する領域のタグ（ｔａｇ）ポインタに検索テーブル２へのタグ（ｔａｇ）ポインタを設定する（Ｓ１０１）。設定する領域は図８のＰ１（命令生成ポインタ）によって指示される。Ｓ１０２、Ｓ１０３、Ｓ１０４、Ｓ１０５がコンパイル処理のメインループであり、バイトコードからネイティブコードを生成する処理である。

【００９２】次に、ＪＩＴコンパイルのコンパイル処理により、バイトコードからネイティブコードへの変換を行う（Ｓ１０２）。変換後のネイティブコードをコードキャッシュ３に格納する前に、命令生成ポインタと解放ポインタの値を比較（命令生成ポインタ＜解放ポインタの条件を満たしているか否かを比較）する（Ｓ１０３）ことで、有効なコードを上書きしてしまわないかチェックする。

【００９３】その結果、命令生成ポインタが解放ポインタより小さければ、全て解放済みの領域なので、コードキャッシュ３にネイティブコードを生成して書き込む（Ｓ１０４）。この時、命令生成ポインタを進める。次に、コンパイル終了条件を満たしたか否かを判断する（Ｓ１０５）。

【００９４】その結果、コンパイル終了条件を満たさなければ、Ｓ１０２の処理へ移行し、コンパイル処理を継続する。また、コンパイル終了条件を満たしたら、生成したネイティブコードを１つのブロックとして構成する。この場合、生成した領域のネクストポインタに、命令生成ポインタの値を設定する（Ｓ１０６）。

【００９５】また、Ｓ１０３の処理で、前記条件を満たしていない場合、すなわち、命令生成ポインタが解放ポインタと等しいか、又は大きい場合、有効なブロックを解放する必要がある。そこで、解放ポインタがコードキャッシュの最後を指している（解放ポインタ＜コードキャッシュの末尾の条件を満たしている）か否かをチェックする（Ｓ１０７）。

【００９６】その結果、解放ポインタがコードキャッシュの最後を指している場合は、解放ポインタをコードキ

ャッシュ３の先頭に設定し（Ｓ１０８）、命令生成ポインタをコードキャッシュ３の先頭のブロックの命令領域に設定する（Ｓ１０９）。この処理により、コードキャッシュ３の先頭に戻ってブロックの解放を行う。

【００９７】次に、解放ポインタが示すタグ（ｔａｇ）ポインタから検索テーブル２のタグ（ｔａｇ）をクリアする（Ｓ１１０）。そして、前記Ｓ１１０の処理により、検索テーブルのタグ（ｔａｇ）が無効化され、解放ポインタが示すブロックが実際に解放される。次に、解放ポインタを次のポインタ（ｎｅｘｔポインタ）に設定して（Ｓ１１１）、Ｓ１０４の処理へ移行する。この処理で、解放ポインタが次のブロックを指すようになる。

【００９８】④：特徴

コードキャッシュから検索テーブルへのポインタを持つことによって、キャッシュブロックのサイズを可変長にでき、メモリを有効に活用することが可能になる。このポインタがない場合でも、検索テーブル全体を調べてキャッシュを無効化することができるが、検索テーブル全体を調べるのは処理量が多く、コンパイルに時間を浪費するが、例９の処理により、キャッシュの無効化を高速に行うことができる。

【００９９】(10)：例１０の詳細な説明

①：例１０の内容

図１０は例１０のコードキャッシュの置換処理説明図である。例１０は、コードキャッシュの置換を行う例であり、コードキャッシュ３に空きがなくなったとき、検索テーブル２の実行カウンタを値の小さいものをスキャンし、実行カウンタの値の小さなものから順に破棄する。実行カウンタはコンパイル直後は適当な大きな値を設定しておき、実行する度に実行したコードに対応するカウンタを増加するようにし、置換が必要となったときに、全体のカウンタを減少するようにする。

【０１００】②：フローチャートによる処理の説明

図１１は例１０の処理フローチャートである。以下、図１１に基づいて例１０の処理を説明する。なお、Ｓ１２１～Ｓ１３６は各処理ステップを示す。また、Ｓ１２１～Ｓ１３１の処理は、図５に示した例２のＳ６１～Ｓ７５の処理と同じである。

【０１０１】インタプリタの処理では、コンパイル済みか否かを判断し（Ｓ１２１）、コンパイル済みでなければ、テーブル登録済み（図４の検索テーブル２のｔａｇに登録されているか否か）を判断する（Ｓ１２２）。その結果、テーブル登録済みでなければ、タグ（ｔａｇ）にアドレス登録を行い（Ｓ１２３）、実行カウンタをクリアする（Ｓ１２４）。

【０１０２】次に、命令フェッチを行い（Ｓ１２５）、デコードを行って（Ｓ１２６）、前記処理と同様にして命令を実行する（Ｓ１２７）。そして、Ｓ１２１の処理に移行する。

【０１０３】また、前記Ｓ１２２の処理でテーブル登録済みであると判断した場合には、検索テーブル２の実行カウンタをＵＰ（＋１）し（Ｓ１２９）、予め決めた特定回数より大きいか否かを判断する（Ｓ１３０）。その結果、特定回数より小さければ、Ｓ１２５に移行しインタプリタで実行し、特定回数以上の場合は、ＪＩＴコンパイラによりバイトコードをコンパイルしてネイティブコードを生成し（Ｓ１３１）、生成したネイティブコードをコードキャッシュ３に登録する（Ｓ１３２）。

【０１０４】また、この時、検索テーブル２の実行カウンタを設定する。次に、前記Ｓ１３２の処理で登録したネイティブコードを実行し（Ｓ１３３）、Ｓ１２１の処理へ移行する。また、Ｓ１２１の処理で、コンパイル済みであると判断した場合は、検索テーブル２の実行カウンタをＵＰ（＋１）して（Ｓ１２８）、Ｓ１３３の処理へ移行する。

【０１０５】一方、前記Ｓ１３１の処理において、ネイティブコードを生成した場合、コードキャッシュ３に空きがあるか否かを判断し（Ｓ１３４）、空きがあれば、Ｓ１３１の処理へ戻る。しかし、コードキャッシュ３に空きがなければ、実行カウンタ全体をスキャンし、個々の実行カウンタを減少させる（Ｓ１３５）。そして、実行回数が少ないエントリを破棄して、コードキャッシュ３に空きを作成し（Ｓ１３６）、Ｓ１３１の処理に戻る。

【０１０６】③：例１０の特徴

頻繁に実行されるネイティブコードが破棄されにくくなる。しかし、空き領域が断片化し、ネイティブコード上で断片化した箇所への無駄な分岐命令が必要となる。

【０１０７】(11)：例１１の詳細な説明

①：例１１の内容

例１１は、コードキャッシュ３のコンパクションの例であり、例１０において、コードキャッシュ３の空き領域の断片化が起きる。この時に、ネイティブコードのかたまりを移動（リロケーション）することで、コンパイラが一度に生成するコードを一塊にまとめる。

【０１０８】②：例１１の特徴

断片化により分断された命令の間に無駄な分岐命令を挿入する必要がなくなる。実行する命令列がコードキャッシュ上に分断されることがなくなり、一塊となるため、命令キャッシュが効率的に使われ、性能が向上する。

【０１０９】(12)：例１２の詳細な説明

①：例１２の内容

例１２は、検索テーブルのヒット率の向上を図る例であり、例５において、コンパイルした全てのバイトコードの命令を検索テーブルに登録せず、コンパイルした命令シーケンスの先頭だけ検索テーブルに登録する。

【０１１０】②：例１２の特徴

検索テーブルのエントリ数にも制限があり、コンパイルした全てのバイトコード命令のアドレスを登録すると、検索テーブル２を上書きする機会が増えるため、コンパ

ーションプログラムの全体の実行速度を速くすることができる。また、前記(2) と比較して、実行カウンタのための領域が必要なく、メモリを節約できる。

【０１２８】(4) ：請求項４では、処理制限手段は、コンパイル時間を短縮するために、コンパイルする範囲を制限し、処理の途中であっても、バイトコードの特定の命令を検出したら、そこまでをコンパイルして処理を中断する命令を生成し、残りはコンパイルしないように処理を制限する。

【０１２９】この場合、従来は、メソッド全体をコンパイルしていたので、全く実行しない部分もコンパイルしていた。これに対して請求項４では、必ず実行する部分だけをコンパイルするので、コンパイルに消費する時間の無駄がなく高速である。

【０１３０】また、従来は、メソッド全体をコンパイルしていたので、全く実行しない部分のネイティブコードも生成していた。しかし、請求項４では、必ず実行する部分だけをコンパイルするので、生成するネイティブコードに無駄がなく、メモリの消費が抑えられる。また、アプリケーションプログラムの応答性を向上させることができる。すなわち、コンパイルしている最中はアプリケーションプログラムの実行は停止するので、この停止している時間が短縮する。

【０１３１】(5) ：請求項５では、第１の破棄手段は、コードキャッシュに空きがなくなった時、コードキャッシュの先頭から順にネイティブコードを破棄する。そして、第２の破棄手段は、ＦＩＬＯ方式で最も昔にコンパイルしたコードから順に破棄する。このようにすれば、ネイティブコードの破棄の方式が簡単なため、実現し易く、ネイティブコードの断片化が起きない。

【図面の簡単な説明】

【図１】本発明の実施の形態における例１の処理フローチャート（その１）である。

【図２】本発明の実施の形態における例１の処理フローチャート（その２）である。

【図３】本発明の実施の形態における例１の検索テーブル例である。

【図４】本発明の実施の形態における例２の検索テーブル例である。

【図５】本発明の実施の形態における例２のコンパイル選択処理フローチャートである。

【図６】本発明の実施の形態における例３のコンパイル選択処理フローチャートである。

【図７】本発明の実施の形態における例４の検索テーブルとコードキャッシュの構成例である。

【図８】本発明の実施の形態における例９のコードキャッシュの置換処理説明図である。

【図９】本発明の実施の形態における例９のコードキャッシュの置換処理フローチャートである。

【図１０】本発明の実施の形態における例１０のコードキャッシュの置換処理説明図である。

【図１１】本発明の実施の形態における例１０の処理フローチャートである。

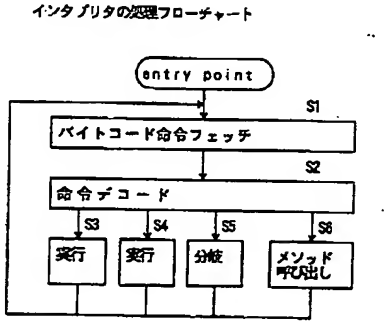
【図１２】従来のインタプリタの処理フローチャートである。

【図１３】従来のインタプリタとＪＩＴコンパイラを組み合わせた時の処理フローチャートである。

【符号の説明】

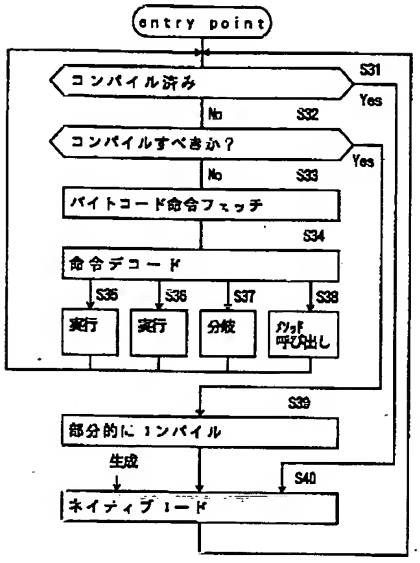
１　コンパイラ
２　検索テーブル
３　コードキャッシュ
４　演算器

【図１２】



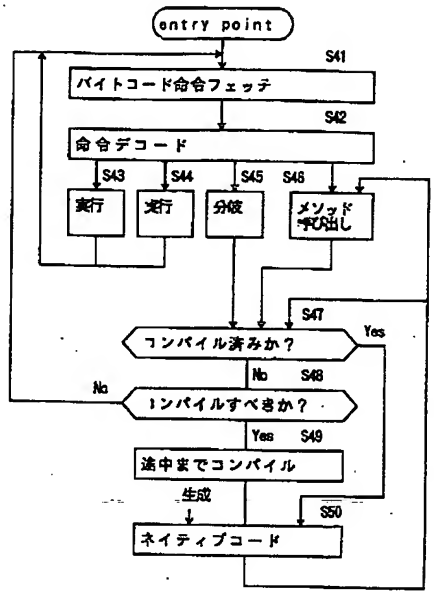
【図1】

例1の処理フローチャート（その1）



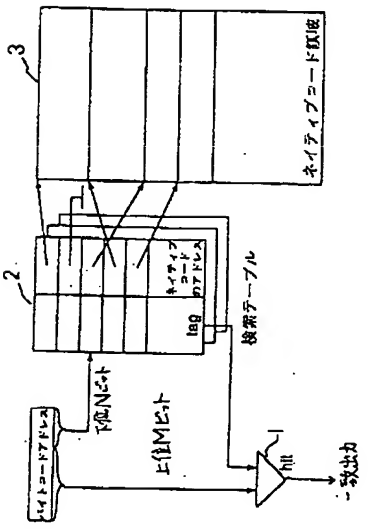
【図2】

例1の処理フローチャート（その2）



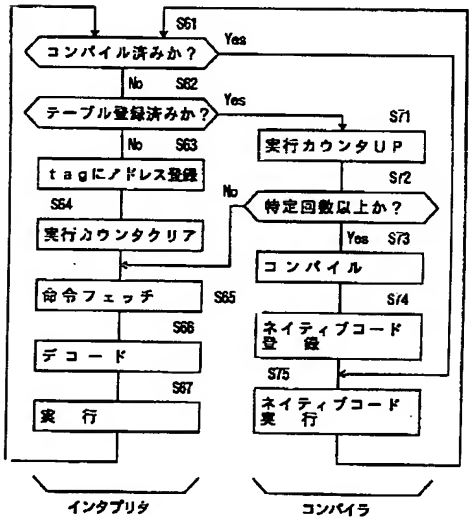
【図3】

例1の検索テーブル例



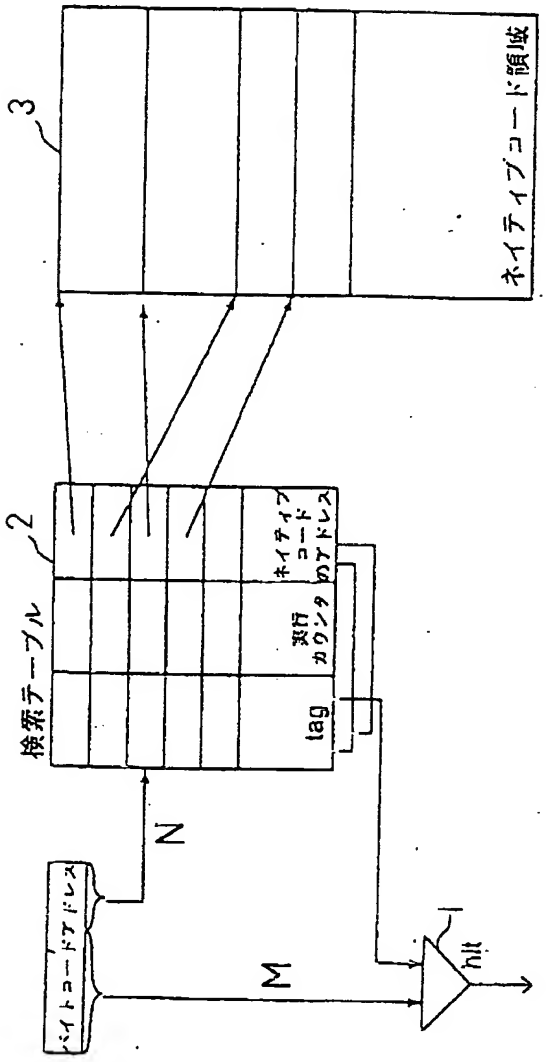
【図5】

例2のコンパイル選択処理フローチャート



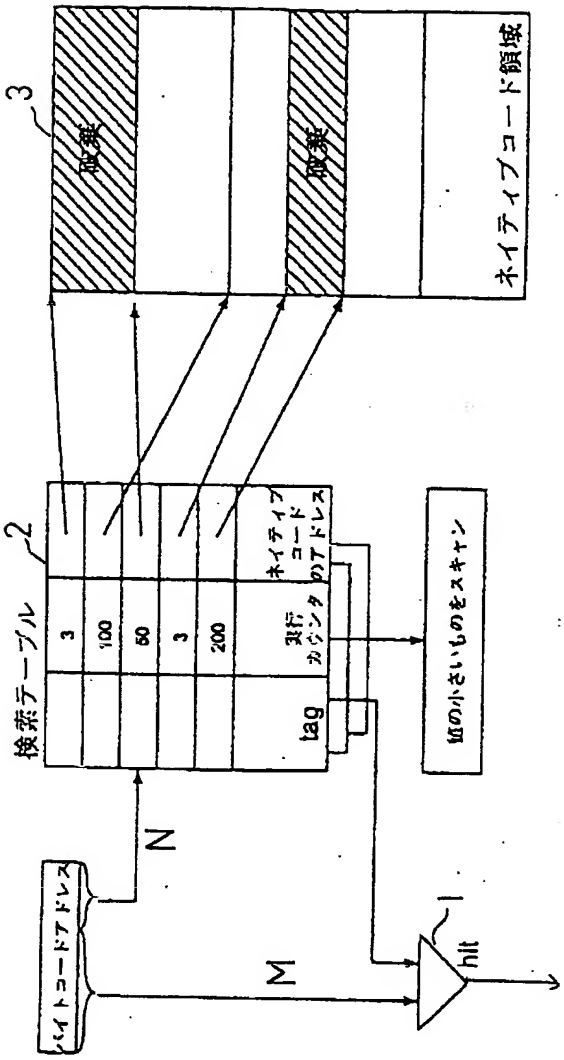
【図4】

例2の検索テーブル例



【図10】

例10のコードキャッシュの置換処理説明図



【図11】

例10の処理フローチャート